

Visual Basic: Objects and collections

Visual Basic is an (OO) *object-oriented* language. Performing a task in Visual Basic (VB) or Visual Basic for Applications (VBA) involves manipulating various types of *objects*, each of which may have several different *properties* and *methods*. To perform a task using VBA you return an *object* that represents the appropriate Excel element and then manipulate it using the objects' *methods* and *properties*.

Objects

A simple statement `Range("A1").Select` illustrates an important characteristic of VB. The syntax of many statements first specifies an *object*, `Range("A1")`, and an action upon it, `Select`. An object is a special type of variable that contains both data and code and represents an element of Excel. Objects exist only in the computer's memory; they don't appear in your code. There are many types of *objects* in VB, and even more in Excel VBA. Excel objects that you will encounter include the `Application` object which represents the Excel application itself, the `Worksheet` object representing a worksheet, and the `Range` object representing an individual cell or a rectangular range of cells (e.g. cells A2:C9 of the currently active worksheet).

Objects have types just as variables have data types.. Object types are called *classes*. `Workbook`, `Worksheet` and `Range` are just a few of Excel's object classes.

For Excel spreadsheets, functions etc. to be controlled and manipulated from a VBA program, these "things" in Excel must have corresponding objects that can be referred to in the program. The collection of objects corresponding to "things" in the Excel application is called the Excel *object model*.

Methods

In VB an action that an object can perform is referred to as a *method*. Thus a *method* of an object is a procedure that carries out some action involving the object. Consider the object `Dog`. To cause it to bark we could write

```
Dog.Bark.
```

However a `Dog` is capable of more than barking, for example we could have

```
Dog.Sit, Dog.Fetch.
```

In Excel, for example, the statement

```
ActiveCell.Delete
```

calls the `Delete` method of the `ActiveCell` object, which deletes the contents of the cell. Like any other procedure, a method of an object can take one or more arguments, as in this example:

```
ActiveCell.AddComment "This is a comment"
```

The list of methods that an object can perform depends on the object. For example, the `Range` object supports about 80 different methods.

A *method* is accessed by following the relevant object with a dot and then the name of the method.

Properties

An *object* can have *properties*. A *property* is a quality or characteristic of the object, e.g. the length of the dog's tail, the loudness of its bark. If you think of *objects* as the nouns of VB, then *properties* are its adjectives and *methods* are its verbs.

In Excel the *properties* may themselves be either primitive data types such as numbers, strings or Boolean values, or may themselves be objects of some kind.

One of the many properties of the `Application` object is called `Name`, and is a read-only string containing the name of the application, i.e. "Microsoft Excel". It can be accessed by adding a dot and the `Name` property after the object:

```
MsgBox Application.Name
```

The `Application` object also has a property called `ActiveCell`, which represents the currently active cell in the active Excel worksheet. `ActiveCell` is an instance of the object type called `Range`, and one of its properties is called `Value` and represents the value (number, string or formula) held by the cell.

The statement

```
Application.ActiveCell.Value = "Hello"
```

will place the string "Hello" in the active cell.

Many properties of the `Application` object can be used without using the qualifier – `Application` in this example – and the above statement could simply be written

```
ActiveCell.Value = "Hello"
```

Let's consider in detail the statement

```
Worksheets("sheet1").Range("A1").Value = 3
```

Step 1 - `Worksheets("sheet1")` evaluates the `worksheets` *method* and returns the *object* that refers to `sheet1`. So we now have

```
[Worksheet object that refers to sheet1].Range("A1").Value = 3
```

Step 2 evaluates the `Range` *method* to return the object that refers to cell A1 of `sheet1`. So we now have

```
[Range object that refers to cell A1 of sheet1].Value = 3.
```

Step 3 evaluates the `Value` *property* of the range object at sets it to the number 3.

Object variables

You have already come across several types of variable in VB, including numerical types (e.g. `Integer`, `Single`, `Double`), the `String` type, and the `Variant` type which can hold a value of any type. There is also another type of variable, called an

Object variable, which can refer to any of the objects in Visual Basic or in the Excel object model.

The syntax for declaring an object variable is much the same as that for any other variable, e.g.

```
Dim rangeA as Range
```

reserves space for a variable that will refer to an object of type Range.

One important difference between an object variable and any other type of variable is that an object variable holds only a *reference* to a specific object, rather than the object itself. (The difference between references and values was discussed in the lecture on modules and procedures, in the context of passing arguments to subroutines and functions.) This distinction may be clearer if we consider a concrete example using primitive and object variables:

```
Dim numA As Integer, numB As Integer
numA = 1
numB = numA
numB = 2
MsgBox ("A=" & numA & ", B=" & numB)
```

numB is a *copy* of numA, so setting numB to have the value 2 has no effect on numA, which still has the value 1.

The situation is different for object variables:

```
Dim fontA As Font, fontB As Font
Set fontA = ActiveSheet.Range("A1").Font
fontA.Bold = False
Set fontB = fontA 'Note: fontB and fontA refer to same object
fontB.Bold = True 'so changing object fontB changes object
                  fontA
```

The object referred to by the variable fontA represents the font used to display the contents of cell A1. fontB is a reference to the same object, not a copy of it, so when the Bold attribute of fontB is changed, fontA is also affected.

You may also have noticed the use of the Set keyword in the above examples. This is another difference between object and primitive variables: when assigning an object reference to an object variable, Set must be used, while it is not needed when assigning a value to a primitive variable.

Collections

A *collection* is an object that contains a group of related objects. Each object within the collection is called an *element* of the collection. Collections are objects so have associated methods and properties.

An example is the `Worksheets` collection, which represents the worksheets in the active workbook. This behaves a bit like an array, in that a specific worksheet in the collection can be referenced using a numeric index:

```
Sheets(2).Activate
```

This makes the second worksheet active. Unlike a normal array, the index in a collection object can be a name instead of a number:

```
Sheets("Chart1").Activate
```

When you want to work with a single object you usually return one member from the collection. The property or method used to return the object is called an *assessor*.

There are some useful ways of dealing with collections built into the VB language. For example, to loop over all the members of a particular collection, one can use the `For Each` syntax:

```
Dim rangeX As Range, cellY As Range
Dim i As Integer
Set rangeX = ActiveSheet.Range("A1:C3")
i = 1
For Each cellY In rangeX.Cells
    cellY.Value = i
    i = i + 1
Next
```

The above piece of code uses a loop, in which the object variable `cellY` refers to each cell member of the collection `rangeX.Cells` in turn. It assigns the value 1 to A1, 2 to A2, 3 to A3, 4 to B1 etc and to 9 to C3. In this case, we could simply have written `rangeX` instead of `rangeX.Cells`, but see what happens if you change it to `For Each cellY In rangeX.Rows`. Now `rangeX` represents the same range of cells in each case, but `rangeX.Cells` and `rangeX.Rows` represent two different collections of objects. In the first case, the members are individual cells, but in the second case the members are ranges representing *rows* of cells, with each row being itself a collection with its own members (the individual cells).

The with statement

The `With` statement provides a way to carry out several operations on the same object with less typing, and often leads to code that is easier to read and understand. For example, instead of

```
Selection.Font.Name = "Times New Roman"
Selection.Font.FontStyle = "Bold"
Selection.Font.Size = 12
Selection.Font.ColorIndex = 3
```

one can write

```
With Selection.Font
    .Name = "Times New Roman"
    .FontStyle = "Bold"
    .Size = 12
    .ColorIndex = 3
End With
```

Using the macro recorder to build an expression

The macro recorder is a convenient way to build expressions that return objects as it knows the object model of the application and the methods and properties of the objects. However it can produce very verbose code. Consider the following example to change the size and font in a chart's title:

```
Sub Macro1()
    ActivateChart.ChartTitle.Select
    With Selection.Font
        .Name = "Times New Roman"
        .FontStyle = "Bold"
        .Size = 24
        .Strikethrough = False
        .Superscript = False
        .Subscript = False
        .OutlineFont = False
        .Shadow = False
        .Underline = False
        .ColorIndex = xlAutomatic
        .Background = xlAutomatic
    End With
End Sub
```

This code contains many redundant lines. Only the size and fontstyle were changed from the default values. This code after recording should be changed to

```
Sub FormatChartTitle()
    With Charts("Chart1").ChartTitle.Font
        .FontStyle = "Bold"
        .Size = 24
    End With
End Sub
```

Setting and getting properties

With some properties you can set and return their values – these are called read-write properties. With some others you can only return their values – these are read-only properties.

We have already met examples of setting a property, e.g.

```
Worksheets("Sheet1").Range("A1").Value = 42
Worksheets("Sheet1").Range("A1").Value = _
    Worksheets("sheet2").Range("B2").Value
```

To get the value property of cell A1 in sheet1 we would use

```
myValue = Worksheets("sheet1").Range("A1").Value
```

There are some properties and methods that are unique to collections. The **Count** property is one. It returns the number of elements in a collection. It is useful if you want to loop over the elements of the collection (though the **For Each ... Next** loop is preferable). The following example uses the Count property to loop over the worksheets in the active workbook, hiding alternate ones:

```

Sub HideEveryOtherSheet()
    For i = 1 To Worksheets.Count
        If i Mod 2 = 0 Then
            Worksheets(i).Visible = False
        End If over collections
    Next i
End Sub

```

Looping on collections

One can loop over collections using variants of the above example. However the recommended way is to use the **For Each ... Next** loop. In this structure VB automatically sets an object variable to return, in turn, each object in the collection. The following code loops over all workbooks open in Excel and closes all except the one containing the procedure:

```

Sub CloseWorkbooks()
    Dim wb As Workbook
    For Each wb In Application.Workbooks
        If wb.Name <> ThisWorkbook.Name Then
            wb.Close
        End If
    Next wb
End Sub

```

The Range object and method

The range object can represent a single cell, a range of cells, an entire row or column even a 3-D range. The range object is unusual in representing both a single cell and multiple cells.

One of the most common ways to return a range object is to use the range method. The argument to the range method is a string, e.g. A1 or a name "myRange".

Examples are:

```

Range("B1").Formula = "=10*RAND()"
Range("C1:E3").Value = 6
Range("A1", "E3").ClearContents
Range("myRange").Font.Bold = True
Set newRange = Range("myRange")

```

The Cells method

The cells method is similar to the range method except that it takes numeric arguments instead of string arguments. When used to return a single cell the first argument is the row, the second is the column. For example, for cell B1,

```

Cells(1,2).Formula = "=10*RAND()"

```

The `Cells` method is particularly useful if you want to refer to cells in a loop using counters. The example loops through the cells A1:D10 and replaces any whose value is less than 0.01 by zero.

```
Sub SetToZero()
    For colIndex = 1 To 4
        For rowIndex = 1 To 10
            If Worksheets("Sheet1").Cells(rowIndex, ColIndex) <_
                0.01 Then
                Worksheets("Sheet1").Cells(rowIndex, _
                    colIndex).Value = 0
            End If
        Next rowIndex
    Next colIndex
End Sub
```

Combining Range and Cell method

You can combine the range and cell methods. Suppose you want to create a range object defined by a top row, a bottom row and left and right columns. The code below returns a range object that refers to cells A1:D10. The `Cells` method defines cells A1 and D10 and the range method then returns the object defined by these cells:

```
Set areaObject = _
    Worksheets("Sheet1").Range(Cells(1,1), Cells(10,4))
```

The Offset method

Sometimes you want to return a range that is a certain number of rows and columns from another cell. The `Offset` method takes an input range object, with `rowoffset` and `columnoffset` arguments to return a range. Suppose the data in a column of cells is either a number or text. The following code writes "Text" or "Number" in the adjacent column.

```
For Each c in Worksheets("sheet1").Range("A1:A10").Cells
    If Application.IsText(c.Value) Then
        c.Offset(0, 1).Formula = "Text"
    ElseIf Application.IsNumber(c.Value) Then
        c.Offset(0, 1).Formula = "Number"
    End If
Next c
```